details, please see “Inside OLE 2, Second Edition, Kraig Brockschmidt, Microsoft Press, 1995” on page 3. So far, ActiveX has been primarily targeted towards the GUI arena although, with COM+, Microsoft has announced it’s intentions to extend it to distributed objects and enterprise components. Then there’s JavaBeans and EnterpriseJavaBeans which targets both the GUI as well as the enterprise/server sections of the markets. And finally, there’s CORBA components - an RFP has been issued by OMG for this and is in the process of standardization. Please refer to “CORBA Components RFP, OMG document #, orbos/96-06-12” on page 3. for details.

We have submitted a response to the CORBA components RFP to deal with enterprise components in the CORBA domain (“CORBA Components, Joint Initial Submission to OMG’s RFP (orbos/96-06-12) by Oracle, IBM, Netscape, SUN and Visigenic et. al. OMG document # orbos/97-11-24.” on page 3).

Elements of the Component Model

What should a standard for component models specify? Here, we outline our notion of what a component model comprises, in terms of high level abstractions. We hope that the workshop will help identify any elements that we have left out and refine the definition of what each of these elements really mean. Any component model - be it for GUIs OR for distributed components - needs to specify the following:

1. *Properties*: An abstract view of the externally visible state of a component.
2. *Constraints* - A means of specifying constraints on property values and cross property constraint specification.
3. *Event Model:* A means of specifying the propogation of occurrences of activity. A specific activity could be a change in the value of a property.
4. *Introspection* - A means of reflecting on the component’s capabilities.
5. *Customization* - A means of customizng a component’s property values.
6. *Composition & containment* - A means of making complex components by putting together simple ones.
7. *Type Aggregation and Interface Navigation* - is the process of putting together interfaces and having a means of navigating between the various ones.
8. *Versioning* - The notion of evolution of interfaces and ways and means of tagging and using these tags.
9. *Packaging & Installation* - Putting a component together for software distribution and then using a component package in building an application

In addition, a “server side” component model will also need to specify:

1. *Transaction Model* - followed by components and responsiblitites of the containers in providing for these services.
2. *Persistence Model* - A model of persisting the state of a component.
3. *Concurrency Control* - Shared access of components.
4. *Security* - It is arguable whether security is a server side consideration alone OR needs to be specified outside of this realm.
5. *Messaging & Qualitites of service specifications for the infrastructure.*

References:

1. JavaBeans Specification, Version 1.01, Graham Hamilton, SUN Microsystems, July 1997
2. CORBA Components RFP, OMG document #, orbos/96-06-12
3. CORBA Components, Joint Initial Submission to OMG’s RFP (orbos/96-06-12) by Oracle, IBM, Netscape, SUN and Visigenic et. al. OMG document # orbos/97-11-24.
4. Inside OLE 2, Second Edition, Kraig Brockschmidt, Microsoft Press, 1995

cific libraries provided prepackaged logic which he could use.

Class libraries provided some set of related classes which one could use as is OR could specialize via inheritance to solve a problem. But they did not deal with designs which would provide solutions to specific problems. Frameworks were proposed as a solution to this - they provided a solution to a problem via an abstract design which one then had to specialize and tune.

The difficulty in using frameworks were that they were abstract reusable “designs” which had to be dealt with by expert programmers to specialize and make concrete. The solution was to combine the advantages of reusable designs with those of reusable implementations. These are components and are the state of the art in software development. A component can be defined to a reusable software artifact which is the unit of software distribution and manageability at design and runtimes. Components are analogous to software ICs and represent a style of programming that is at a much higher level of abstraction that anything that is seen today.

Why Use Components for Software Development?

Components raise the level of abstraction to that which can be easily used by a domain expert who is not necessarily an expert programmer. They allow software vendors to build visual development environments in which the concept of plugging together these “software ICs” forms the basis of any new development. The writing of actual code is kept to a minimum - scripting can be used to glue together components or to tailor existing behavior. A typical development effort using components would be importing the components of interest and customizing each one them WITHOUT explicit coding and finally wiring together the components to form an application. The advantages are immediately obvious:

- Increased productivity gained by reuse of design and implementation
- Increased reliability by using well tested code
- Lower maintainence costs because of a smaller code base.
- Minimizes effects of change since black box programming tends to rely on interfaces as compared to explicit programming.
- Components provide a well encapsulated mechanism to package, distribute and reuse software.

Barriers to Reuse

Although there are several known reasons for lack of reuse, the primary one which we wish to focus on is the lack of standards in software development. We will further limit our discussions to component based development since it is our premise that all software will be developed using components for the reasons mentioned above. There are several vendors who sell environments to support component based development. Since there is no interoperability between vendor’s products (for lack of a standard), no developer can expect to write components that will run seamlessly in different vendor’s environments. This is the primary barrier to reuse of components and open, component based software development. JavaBeans has become a defacto standard for Java components as is ActiveX/COM/COM+ .There is however, no such standard for components written in other languages OR for other platforms. What is required is a single standard for components that will be able to seamlessly interoperate with the existing ones of JavaBeans and COM+. A user living in an environment which supports all of these standards should be able to pick up a component, be it a JavaBean, a COM component etc, and use it in the paradigm that he is programming in without any interference from him. To clarify this, if the user is developing using the JavaBean paradigm, the component he picks up should be available *as* a JavaBean in that environment as far as the API are concerned. This kind of standards based interoperability is the key to large scale reuse.

Standards for Software Components

What are the actual standards for component (based) development available in this area? There’s the COM+ standard which is defined by Microsoft and is available on only one platform - Windows. For

The Role of Components & Standards in Software Reuse

Dr. Umesh Bellur
Oracle Corporation
ubellur@us.oracle.com

Introduction

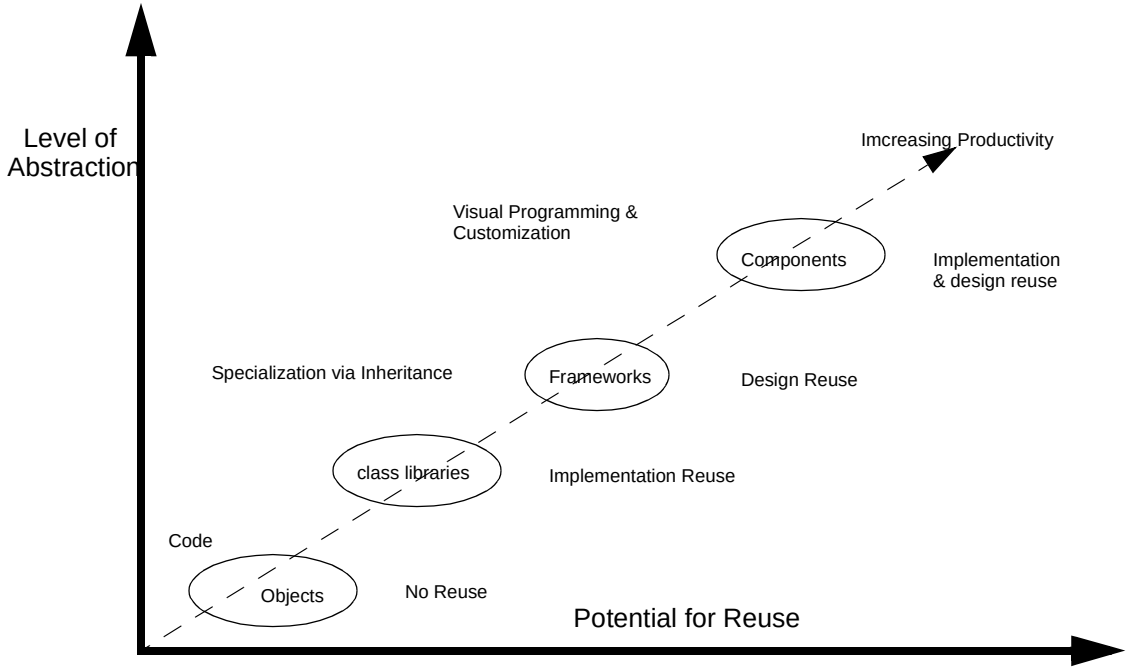
This paper presents the role of standards as we see it in the use of and acceptance of components for the rapid development and deployment of enterprise software applications. Our thesis is that the existence of standards (defacto or otherwise) and adherence to them to promote open systems is the primary motivation to build reusable software and well as reuse it. Standards ensure consistency, compatibility and allow multiple vendor's products to interoperate thereby fostering resuse.

The views we have proposed here stems from our experience in putting together a proposal for an OMG standard CORBA component model (jointly with with SUN, Netscape, IBM, Visigenic & others) as well as from internal development efforts to support an open architecture for development of distributed applications using components.

Evolution of Abstraction & Reuse

Aeons ago, people used assembly langauges to program computers. As we grew out of the assembly ages, along came high level languages which made the expression of ideas simpler and more compact. As programmers discovered that they spent time reprogramming existing data-structures, class libraries were born. This raised both the level of abstraction as well as the level of reuse in application development. Then came object orientation. The notions of encapsulation, information hiding and polymorphism proved irresistible. Modular development was reborn with object orientation's renewed popularity in the early to mid 80s.

But merely programming with objects did not ensure reuse. It was only after the notion of objects was



extended to libraries of classes, that reuse began to take serious shape. Several commercial class libraries began to appear and the popularity of these libraries focused people's attention on the promise afforded by the reuse of software. At the same time, the level of abstraction which was afforded to the applications developer was being raised from having to program all his objects to one where generic and domain spe-